Deep Equilibrium Models

Shaojie Bai, J. Zico Kolter, Vladlen Koltun

Presented by: Alex Wang, Gary Leung, Sasha Doubov
Agenda

1. Weight tying and infinite depth models
2. Implicit layer formulation
3. Approximation and computational considerations
4. DEQ stacking?
5. Experiments
Motivation

Let’s start with a typical deep NN architecture:

\[
\begin{align*}
    z_1 &= x \\
    z_{i+1} &= \sigma(W_i z_i + b_i), \quad i = 1, \ldots, k - 1 \\
    h(x) &= W_k z_k + b_k
\end{align*}
\]

Image courtesy of Deep Implicit Layers Tutorial
Motivation

**Weight-Tying:** Use the same $W$ and inject the input for each layer

\[ z_1 = 0 \]
\[ z_{i+1} = \sigma(W z_i + U x + b), \quad i = 1, \ldots, k - 1 \]
\[ h(x) = W_k z_k + b_k \]

Image courtesy of Deep Implicit Layers Tutorial
Motivation

**Weight-Tying:** Use the same $W$ and inject the input for each layer

Focusing on activation iteration:

$$z_{i+1} = \sigma(Wz_i + Ux + b), \ i = 1, \ldots, k - 1$$

[Image courtesy of Deep Implicit Layers Tutorial]
In the infinite limit, as $i \to \infty$ (under nice conditions)

**Key insight:** The network’s activations $z^*$ approach a fixed point!

This is a DEQ!

$$z^* = \sigma(Wz^* + Ux + b), \ i \to \infty$$
Deep Equilibrium Model Overview

Diagram showing the comparison between a typical deep neural network and a deep equilibrium model. The diagram highlights the key difference in memory storage needed at training time. The deep equilibrium model avoids the need for memory storage as $L \to \infty$. The equilibrium solver for the deep equilibrium model is also shown, with $z^* = f_\theta(z^*; x)$.
Implicit vs. Explicit Layers

Explicit Layers:  
Typical Neural Network layers, which can directly compute the output and backward pass through backprop

Implicit Layers:  
Based on solving solution to some problem, such that $x, z$ satisfy some condition

- Arises in naturally in some domains, such as ODEs and fixed-points

Image courtesy of Deep Implicit Layers Tutorial
Forward Pass

**Naive Approach:** we could repeatedly apply the function until convergence

\[ z^{[i+1]} = f_{\theta}(z^{[i]}, x) \quad \text{for } i = 0, 1, 2, \ldots \]

**Better Way:** Use a root-finding algorithm to find the fixed point

1. Reformulate fixed-point as finding the root:
   \[ g_{\theta}(z^*, x) = f_{\theta}(z^*, x) - z^* \rightarrow 0 \]
   We’ll use this notation from here on!

2. Apply generic root-finding algorithm (ex. Newton’s method!)
   \[ z^* = \text{RootFind}(g_{\theta}; x) \]
Backward Pass

We need to update the parameters $\Theta$ in our model, to minimize our loss function

\[
\frac{\partial z^*}{\partial (\cdot)}
\]

**Challenge:** differentiating through fixed point

**Naive Approach:** Built-in Autodiff, through solver computation graph

- Memory Issues
- Floating Point Errors

**Better Approach:** Use Implicit Function Theorem!
Implicit Function Theorem (Informal)

Let \( f \) be a relation with inputs \( x_0, z_0 \)

1. \( f(x_0, z_0) = 0 \)
2. \( f \) is continuously differentiable with non-singular Jacobian
   \[ \partial_1 f(x_0, z_0) \in \mathbb{R}^{n \times n} \]

Then there exist neighborhoods (open sets) \( S_{x_0}, S_{z_0} \)
around \( x_0 \) & \( z_0 \) and a function \( z^* : S_{x_0} \to S_{z_0} \)

1. \( z_0 = z^*(x_0) \)
2. \( f(x, z^*(x)) = 0 \quad \forall x \in S_{x_0} \)
3. \( z^* \) is differentiable on \( S_{x_0} \)
Implicit Function Theorem

High-level Idea:

- Convert a relation to a function in a local region and find its derivative
  \[ f(x, z) = x^2 + z^2 - 1 = 0 \]
- Explicit function at A:
  \[ g_A(x) = \sqrt{1 - x^2} \]
- IFT allows us to find the derivative of \( g \), without the explicit form
Implicit Function Theorem

Let $f : \mathbb{R}^p \times \mathbb{R}^n \to \mathbb{R}^n$ and $x_0 \in \mathbb{R}^p$, $z_0 \in \mathbb{R}^n$ be such that:

1. $f(x_0, z_0) = 0$
2. $f$ is continuously differentiable with non-singular Jacobian $\partial_1 f(x_0, z_0) \in \mathbb{R}^{n \times n}$

Then there exist open sets $S_{x_0} \subset \mathbb{R}^p$ and $S_{z_0} \subset \mathbb{R}^n$ containing $x_0$ and $z_0$ respectively, and a unique continuous function $z^* : S_{x_0} \to S_{z_0}$ such that

1. $z_0 = z^*(x_0)$
2. $f(x, z^*(x)) = 0 \quad \forall x \in S_{x_0}$
3. $z^*$ is differentiable on $S_{x_0}$

\[ f(x, z) = x^2 + z^2 - 1 = 0 \]
Backwards Pass

\[ z_0 = f(x_0, z_0) \]

By IFT

\[ z^*(x) = f(x, z^*(x)) \quad \forall x \in S_{x_0} \]

\[ \partial z^*(x_0) = \partial_0 f(x_0, z_0) + \partial_1 f(x_0, z_0) \partial z^*(x_0) \]

\[ \partial z^*(x_0) = [I - \partial_1 f(x_0, z_0)]^{-1} \partial_0 f(x_0, z_0). \]
Backwards Pass - DEQ

- **Backward Pass:**
  - Solve using root finding (e.g. Newtons)

\[
\frac{\partial \ell}{\partial (\cdot)} = - \frac{\partial \ell}{\partial z^*} (J_{g_\theta}^{-1}|_{z^*}) \frac{\partial f_\theta(z^*; x)}{\partial (\cdot)}
\]

\[
(J_{g_\theta}^\top|_{z^*}) x^\top + \left( \frac{\partial \ell}{\partial z^*} \right)^\top = 0
\]

\[
g_\theta(z^*; x) = f_\theta(z^*; x) - z^*
\]

VJP easily obtained from Pytorch/Jax/etc.
Approximate Inverse Jacobian - Broyden’s Method

- Expensive to calculate the inverse Jacobian during root finding for both forwards and backwards!

- **Broyden’s Method (quasi-newton solver):**
  - During root finding, approximates the inverse Jacobian using

\[
J_{g_\theta}^{-1} \big|_{z[i+1]} \approx B_{g_\theta}^{[i+1]} = B_{g_\theta}^{[i]} + \frac{\Delta z^{[i+1]} - B_{g_\theta}^{[i]} \Delta g_\theta^{[i+1]}}{\Delta z^{[i+1]^T} B_{g_\theta}^{[i]} \Delta g_\theta^{[i+1]}} \Delta z^{[i+1]^T} B_{g_\theta}^{[i]}
\]

**Initial Guess:** \( B_{g_\theta}^{[0]} = -I \)

\[g_\theta(z^*; x) = f_\theta(z^*; x) - z^*\]
DEQ Memory

- **Very memory efficient** because forward and backward passes just use root-finding algorithms.
- Avoids over all the overhead from uncurling backpropagating steps.
- **Storage:**
  - Equilibrium Point $z^*$
  - Network Input $\mathcal{X}$
  - Model $f_\theta$
  - VJP (*no Jacobian construction needed!*)
Expressivity of DEQs

Intuition:
Consider a simple function composition
Transforming this into a DEQ:
\[ y = g_2(g_1(x)) \]

\[ f(z, x) = f\left(\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}, x\right) = \begin{bmatrix} g_1(x) \\ g_2(z_1) \end{bmatrix} \]

Thus, the equilibrium point is:
\[ z^* = f(z^*, x) \iff z_1^* = g_1(x), \quad z_2^* = g_2(z_1^*) = g_2(g_1(x)) \]

The output equilibrium is the output of the function!
*(can be extended to arbitrary computation graph)*
DEQ Stacking?

- Stacking DEQs don’t really work, as a single DEQ layer can model any amount of stacked DEQ layers.

Intuition:

Consider a stack of 2 DEQ layers:

\[
\begin{align*}
z_1^* &= f_1(z_1^*, x) \\
\end{align*}
\]

\[
\begin{align*}
\rightarrow z_2^* &= f(z_2^*, z_1^*)
\end{align*}
\]

This is equivalent to the following single equilibrium problem:

\[
\begin{align*}
\begin{bmatrix}
z_1^* \\
z_2^*
\end{bmatrix} &= \begin{bmatrix}
  f_1(z^*, x) \\
  f_2(z_2^*, z_1^*)
\end{bmatrix} = f(z^*, x)
\end{align*}
\]
Experiments

- Apply Deep Equilibrium Networks to sequence modelling tasks
  - Sequence empirically converge
  - Already use of weight tying over the temporal sequence (Trellis Nets & Transformer models)
  - Long-range copy-memory, Penn Treebank Language Modelling, WikiText-103

- Demonstrate the memory efficiency and expressivity of DEQ models given similar parameter counts as well as the speed of computation
Convergence Caveat

- One might expect the network to diverge in the infinite limit.
- In practice, many networks do not, which is explored more formally in a later work [Kolter et. al 2020].
- In this work, the authors empirically show that the contributions of subsequent layers diminish at very large depths.
# Experiments

## Word-level Language Modeling w/ WikiText-103 (WT103)

<table>
<thead>
<tr>
<th>Model</th>
<th># Params</th>
<th>Non-Embedding Model Size</th>
<th>Test perplexity</th>
<th>Memory†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic TCN [7]</td>
<td>150M</td>
<td>34M</td>
<td>45.2</td>
<td>-</td>
</tr>
<tr>
<td>Gated Linear ConvNet [17]</td>
<td>230M</td>
<td>-</td>
<td>37.2</td>
<td>-</td>
</tr>
<tr>
<td>AWD-QRNN [33]</td>
<td>159M</td>
<td>51M</td>
<td>33.0</td>
<td>7.1GB</td>
</tr>
<tr>
<td>Relational Memory Core [40]</td>
<td>195M</td>
<td>60M</td>
<td>31.6</td>
<td>-</td>
</tr>
<tr>
<td>Transformer-XL (X-large, adaptive embed., on TPU) [16]</td>
<td>257M</td>
<td>224M</td>
<td><strong>18.7</strong></td>
<td>12.0GB</td>
</tr>
<tr>
<td>70-layer TrellisNet (+ auxiliary loss, etc.) [8]</td>
<td>180M</td>
<td>45M</td>
<td>29.2</td>
<td>24.7GB</td>
</tr>
<tr>
<td>70-layer TrellisNet with gradient checkpointing</td>
<td>180M</td>
<td>45M</td>
<td>29.2</td>
<td>5.2GB</td>
</tr>
<tr>
<td>DEQ-TrellisNet (ours)</td>
<td>180M</td>
<td>45M</td>
<td><strong>29.0</strong></td>
<td><strong>3.3GB</strong></td>
</tr>
<tr>
<td>Transformer-XL (medium, 16 layers)</td>
<td>165M</td>
<td>44M</td>
<td>24.3</td>
<td>8.5GB</td>
</tr>
<tr>
<td>DEQ-Transformer (medium, ours).</td>
<td>172M</td>
<td>43M</td>
<td>24.2</td>
<td><strong>2.7GB</strong></td>
</tr>
<tr>
<td>Transformer-XL (medium, 18 layers, adaptive embed.)</td>
<td>110M</td>
<td>72M</td>
<td>23.6</td>
<td>9.0GB</td>
</tr>
<tr>
<td>DEQ-Transformer (medium, adaptive embed., ours)</td>
<td>110M</td>
<td>70M</td>
<td><strong>23.2</strong></td>
<td>3.7GB</td>
</tr>
<tr>
<td>Transformer-XL (small, 4 layers)</td>
<td>139M</td>
<td>4.9M</td>
<td>35.8</td>
<td>4.8GB</td>
</tr>
<tr>
<td>Transformer-XL (small, weight-tied 16 layers)</td>
<td>138M</td>
<td>4.5M</td>
<td>34.9</td>
<td>6.8GB</td>
</tr>
<tr>
<td>DEQ-Transformer (small, ours).</td>
<td>138M</td>
<td>4.5M</td>
<td><strong>32.4</strong></td>
<td><strong>1.1GB</strong></td>
</tr>
</tbody>
</table>

Not SOTA, but good at same param size

Efficient!
Experiments - Runtime

Table 4: Runtime ratios between DEQs and corresponding deep networks at training and inference (> 1× implies DEQ is slower). The ratios are benchmarked on WikiText-103.

<table>
<thead>
<tr>
<th>DEQ / 18-layer Transformer</th>
<th>DEQ / 70-layer TrellisNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>Inference</td>
</tr>
<tr>
<td>2.82×</td>
<td>1.76×</td>
</tr>
</tbody>
</table>

- Notice DEQs are slower!
- This is a consequence of solving an inner optimization inside the network
Conclusion

● Deep Equilibrium Models are a weight tied, approximately infinite depth neural network
  ○ Output is the fixed point of some neural network function

● Computes two root finding solutions for both the forward and backwards pass
  ○ Uses IFT to compute the gradient updates rather than backpropagation and autodiff through the iterative graph

● Performs comparatively to SOTA models of the same size but are considerably more memory efficient
  ○ Typically slower due to the inner loop optimization both forward and backwards
References
